A Novel Method for Online Detection of Faults Affecting Execution-Time in Multicore-Based Systems

A novel method for online detection of faults affecting execution-time in multicore-based systems

STEFANO ESPOSITO, Politecnico di Torino
MASSIMO VIOLANTE, Politecnico di Torino
MARCO SOZZI, Leonardo-Finmeccanica
MARCO TERRONE, Leonardo-Finmeccanica
MASSIMO TRAVERSONE, Leonardo-Finmeccanica

This paper proposes a bounded interference method, based on statistical evaluations, for online detection and tolerance of any fault capable of causing a deadline miss. The proposed method requires data that can be gathered during the profiling and worst-case execution time (WCET) analysis phase. This paper describes the method, its application, and then it presents an avionic mixed-criticality use case for experimental evaluation, considering both dual-core and quad-core platforms. Results show that faults that can cause a timing violation are correctly identified; other faults that do not introduce a significant temporal interference can be tolerated to avoid high recovery overheads.

• Computer Systems Organization→Dependable and fault-tolerant systems and networks

• Applied computing→Physical sciences and engineering→Aerospace.

Additional Key Words and Phrases: software implemented hardware fault tolerance; mixed-criticality applications; safety-critical systems; hard real-time applications; multicore-based systems.

ACM Reference Format:

1. INTRODUCTION

The last decade has been characterized, for what concerns computer engineering, by the rise of multicore architectures. Starting in the second half of the 2000s, many multicore architectures have been released in all market sectors, highly improving the computational capabilities of the products in which they have been deployed. These products, however, have been mostly used in the consumer electronics market, with embedded systems featuring multicore chips included in many new high-end domestic appliances, infotainment systems, and likewise. However, there are several industries which have not yet been able to use multicore chips to their full extent. Such industries are those facing strict certification and/or qualification processes, as they are mainly dealing with safety critical hard real-time systems. Such industries as avionics, space, defense, automotive, and railways signaling have not yet been able to fully exploit the potential improvements offered by multicore architectures. The struggle of cited

This research was partially supported by the ECSEL Joint Undertaking project in the Innovation Pilot Programme "Computing platforms for embedded systems" (AIPP5) under grant agreement n. 621429 (project EMC²).

Author's addresses: S. Esposito and M. Violante, Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy; M. Sozzi and M. Traversone, Leonardo-Finmeccanica, Nerviano (MI), Italy; M. Terrone, Leonardo-Finmeccanica, Torino, Italy.

industries, when using multicore architectures, is mainly due to interferences between tasks running on the different cores within the same processor. The main concern, when facing certification and qualification, is the unpredictable interference in performance introduced by the sharing of key system resources among the cores of a multicore processor. Such resources usually include the level-2 (L2) cache and the system bus, the main memory, and some peripherals such as sensors and actuators. Sharing these resources could have heavy impact on the worst case execution time (WCET), making it very difficult to guarantee the feasibility of scheduling schemes for the system under analysis.

Multicore architectures have been optimized in the years for the symmetric multi-processing (SMP) scenario, in which several threads of the same application run in parallel on different cores and can migrate seamlessly from one core to another. In this scenario, the fairness in the access to the shared resources is important, since all cores perform the same work. However, the same fairness is not so advisable when a different scenario is considered. The most interesting application scenario for the industries cited above is the mixed-criticality scenario, in which tasks with different criticalities, i.e. different levels of assurance in a certification context, are executed on the same hardware. The work proposed in this paper starts from the findings of the many papers already published on the topic to propose a novel method for detecting interferences among cores to tolerate misbehaviors of tasks sharing the same multicore processor with other tasks by means of graceful degradation. The detection is based on hardware readily available in the multicore architectures and accessible through assembly instructions. Using such hardware, it is possible to detect when a monitored task is suffering from interferences by other tasks and to react to mitigate the effects of such interferences. Graceful degradation aims at reducing the level of service of the system while keeping it in a safe state. In this paper, we will describe how the hardware is used to achieve such results and how scheduling algorithms can be modified to take into account such measures. The method proposed in this paper can be considered as a bounded interference method, in which the shared resource is the memory hierarchy. The same method can be easily implemented considering different shared resources, provided that a suitable performance counter exists in the system. We will describe how the method has been implemented on a benchmark mixed-criticality application and evaluated.

The rest of the paper is organized as follows: Section 2 describes the most relevant works already published on the topic; Section 3 describes in detail the proposed method and its theoretical foundations; Section 4 describes how the method is applied to a benchmark application; Section 5 discuss the experimental evaluation and its results; Section 6 draws some conclusions and outlines possible future works to further integrate the proposed method in state-of-the-art system architectures to achieve an overall fault-tolerance suitable for most industries.

2. RELATED WORKS

In this section an overview of related works on the topic of multicore systems is presented.

Many industries have been looking at multicore systems as a way of reducing space, weight and power consumption (SWaP) of electronic equipment. Many industries are already using applications composed of several tasks sharing the same hardware in production systems, although not using multicore architectures. Here we summarize the most relevant researches for the scope of this paper. The interested reader may

find a more comprehensive review of the major topics in in Paulitsch et al. [2015] and in Burns and Davis [2016].

The most important approach in consolidation of multiple applications is partitioning. In Rushby [1999], partitioning is defined as "appropriate hardware and software mechanisms to restore strong fault containment". Partitioning can be categorized as spatial and temporal partitioning:

- *Spatial partitioning* is implemented by means of memory management units (MMUs) and memory protection units (MPUs). Recently, type-1 hypervisors, i.e. virtualization software that does not rely on the services of an operating system, have implemented partitioning of resources by means of MMU. The resources are memory areas which can represent either memory-mapped hardware peripherals, or actual instruction and data memory. In Avramenko et al. [2015], a type-1 hypervisor has been used to implement a resource partitioning on a dual-core microprocessor. The system was designed to implement isolation of two applications running at different levels of assurance, and fault tolerance for both hardware and software faults.
- *Temporal partitioning* is the most critical when considering multicore applications. In multicore processors, shared resources are intrinsic and the effect of sharing is hard to asses without very detailed information on the architecture. Such information is usually very hard to come by for end-users and original equipment manufacturers (OEMs), which usually buy commercial-off-the-shelf (COTS) systems and integrate their own hardware and software with them. Such considerations explain why it is so difficult to prove real absence of interference, which in turn explains why multicore architectures are not so widespread in safety-critical systems.

The certification authorities software team (CAST) produced a position paper [CAST 2014], describing how software running on a dual-core system should be implemented in order to achieve a safe behavior. The document identifies several problem areas and it proposes steps to solve the issues and to adopt multicore architectures in avionic systems. Today, most solutions for temporal interference can be categorized into time multiplexing approaches, and bounded interference approaches, as suggested in Paulitsch et al. [2015]. *Time multiplexing* solutions resort to an offline analysis of the system, in order to schedule the tasks in a way that reduces interference. Many approaches split the applications in phases, each phase can or cannot access shared resources. Phases that can access resources are never scheduled in parallel on different cores, i.e., while one core is running a phase containing accesses to shared resources, the other cores must be running phases that only use local resources [Schranzhofer et al. 2009; Boniol et al. 2012]. Different approaches can add some other requirements and further categorize the phases, for instance Pellizzoni et al. [2011] identify phases in which no access to shared resources can be performed nor any code external to the application should be executed; for instance, in such phases no system call or interrupt service routine can be executed. However, all time multiplexing approach share the defect of potentially wasting a high amount of computational power, since they all introduce some idle time. This can be explained looking at Fig. 1, which represents a time diagram of the scheduling of four tasks on four cores with a time multiplexing approach. In the considered scenario, the shared resource is the memory. Tasks are split into two phases: a memory access phase and a computation phase. The memory access phase performs read and write operations in the shared memory, while the computation phase uses only local resources, e.g. L1 cache, to perform its operations. Fig. 1 shows how the time multiplexing approach introduces long idle periods.

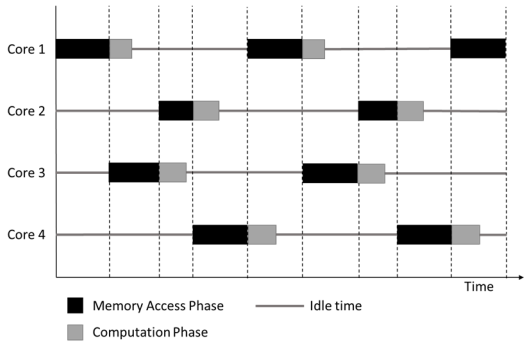

Fig. 1. Time diagram showing four tasks on as many cores. The scheduling uses a time multiplexing approach for accesses to the shared memory. Each time a core is accessing the memory, all other cores can only execute using local resources, e.g. private caches. The time diagram shows that this approach introduces long idle periods.

Bounded interference approaches describe methods to evaluate a maximum interference between different applications on a multicore system. Interferences are evaluated in the worst-case scenario, thus bounded interference approaches can also introduce idle periods. However, there are some approaches that dynamically assign those idle periods to other tasks, thus reaching a very high usage of computational resources when compared to time multiplexing solutions. An extension of WCET analysis is proposed in Nowotsch et al. [2014a], where the delay in accessing resources is evaluated depending on how many concurrent accesses there might be. This WCET analysis methodology is named interference sensitive WCET (isWCET). Fig. 2 represents the application of this method to a quad-core architecture.
The approach can be extended as described in Nowotsch and Paulitsch [2013], by implementing a method for assigning idle time. The analysis described in Fig. 2 allows for some waste in computational resource, since not all concurrent accesses will result in the maximum delay possible. A safety-net using performance counters is implemented, which traces the accesses to the shared resource. When a task completes its accesses, the elapsed time is compared to the worst-case time computed offline and the remaining time, if any, can be used for additional computation. The safety-net can also be used to detect errors induced by single event upsets (SEUs) or other external perturbation of the system. If a task exceeds the maximum number of accesses that it should perform, all accesses to that resource by that task are blocked.

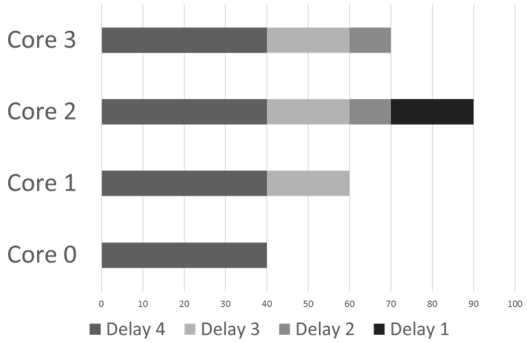

Fig. 2. isWCET applied to a quad-core system running four tasks. The first forty accesses on each core are accounted as introducing the maximum delay possible, i.e. the one experienced when all four cores try a concurrent access to the shared memory hierarchy. The next twenty are accounted as introducing the same delay experienced when three cores attempt a concurrent access. The next ten accesses are accounted as introducing the same delay introduced by two concurrent accesses. Finally, the last twenty accesses are accounted as introducing the nominal delay for an access to the shared memory hierarchy.

Many approaches for mixed criticality applications are based on a scheduling algorithm. For instance, Vestal [2007] describes a schedulability test and a priority assignment scheme based on said test for mixed-criticalities applications. The approach was extended in Baruah and Vestal [2008] to any number of assurance levels, whereas Vestal [2007] considered a given set of assurance levels. Anderson et al. [2009] further extended the algorithm to apply it to a multicore scenario, and many papers since then have proposed scheduling algorithms for mixed-criticality applications on multicore systems. Further scheduling approaches have been proposed also with fault tolerance schemes integrated in the scheduling algorithm [Giannopoulou et al. 2013; Giannopoulou et al. 2014; Krishna 2014; Pathan 2014]. In Nowotsch et al. [2014b] a method is proposed for monitoring execution time in multicore-based system-on-chips (SoCs). The method is based on an offline analysis of the WCET and of the resource accesses, and on an online performance counter usage. This paper proposes a similar method, but describes and justifies in more detail the approach. Furthermore, the main novelty of the approach proposed in this paper is the introduction of an online fault-detection mechanism, i.e. a way of detecting anomalies due to hardware or software faults which cannot be accounted for in a purely offline analysis. Moreover, this paper proposes an experimental evaluation of the approach, which is lacking in Nowotsch et al. [2014]. The proposed method improves the idea presented in Esposito et al. [2016] by extending and justifying it, in order to improve detection rate and allow the introduction of a more fine-grained recovery strategy.

3. INTERFERENCE DETECTION BASED ON PERFORMANCE MONITORS

This paper proposes a method for online detection of interference between tasks running on a multicore-based system-on-chip (SoC). The method is completely independent on the underlying scheduling approach, moreover it does not require any knowledge on the delay of access to the shared resource, nor any modification to the WCET analysis approach.

The approach proposed in this paper can be classified as a bounded interference approach. It is composed of two phases, one offline, the other online. The offline phase serves the purpose of bounding the usage of a shared resource by a given task. The online phase implements interference detection and recovery. For the offline phase, we propose the use of profiling data available from design activities, such as WCET estimation, to compute three thresholds to be used during the online phase. The online

phase can be classified as a safety-net as defined in Paulitsch et al. [2015]: it uses performance counters to evaluate the use of a shared resource by a monitored task.
The proposed method can be implemented to work with two granularity levels:

1. *Core granularity* is the coarsest of the two. It requires all tasks of interest to be associated to a single core. The profiling and analysis described in this section should be then performed on that core.
2. *Task granularity* requires the profiling and analysis described in this section to be performed on each task of interest. It also requires a support from the operating system: the performance counters should be part of the task context to be saved and restored at context-switch.

In this section we describe the method in its theoretical aspects. In Section 4 a practical application of this method to a benchmark application is proposed.

3.1 Metric definition

In the scope of this paper, we refer to the concept of interference as any phenomenon that is able to induce a misbehavior in a monitored task. Such a misbehavior, if unchecked, can cause a failure in the system. This paper is focused on *temporal interferences*, i.e. those interferences that cause a longer-than-expected execution time.
An interference metric, henceforth also simply *metric*, is any quantity that can be measured at runtime and that changes when the monitored task is affected by an interference.
Two or more interference metrics can be aggregated in any convenient way, e.g. using a sum. In the scope of this paper, the result of such an aggregation is a *compound metric,* whereas the aggregated metrics are called *component metrics.*
A *well-defined* compound metric is a compound metric that satisfies the properties of an interference metric: it has to be measurable at runtime, and it has to change when the monitored task is affected by an interference. For a compound metric to be measurable at runtime, all of its component metrics must be measurable at runtime. For a compound metric to change when the monitored task is affected by an interference, at least one of its component metrics should change when the monitored task is affected by an interference.

3.2 Metric selection

To be able to use a metric, it is necessary to identify a suitable measuring tool, such as specialized hardware purposefully added to the system, or hardware already implemented for other purposes. One example of such hardware is the performance counters available in most multicore-based SoCs.
The main purpose of performance counters is to be used in the profiling and debugging phases of development, although they can be used as a safety-net in a scheduling approach. The metrics that can be measured by performance counters depend on the hardware architecture [ARM 2012a, NXP 2014], but they usually include:

- **Cycle counters:** these counters are incremented at each clock cycle of the processor, or each n clock cycles, where n is usually configurable.
- **Cache hit/miss counters:** these counters are incremented each time a cache hit/miss occurs. Most architectures have different counters for instruction and data caches.
- **Data read/writes:** these counters are incremented each time a read or write operation in data memory is issued.
- **Branch prediction miss:** this counter is incremented each time a branch prediction was wrong or a branch was not predicted.

- **Exception taken/return:** the exception taken counter is incremented each time an exception or an interrupt are serviced; the exception return counter is incremented each time an exception handler or interrupt service routine returns.

Any of the quantities measured by the performance counters can be used as interference metric, provided that their value changes when the monitored task is affected by an interference. The designer should also select the most relevant metric to its application.

When an interference is considered, the monitored task can be either a *victim*, or an *offender*. A victim is a task that is affected by an interference generated by a different task. An offender is a task that is generating an interference. When selecting a metric, the designer should take into account the difference between victim and offender and he/she should select the proper metric to identify the monitored task as either a victim or an offender. For instance, an offender task could be detected because its accesses to the shared resources are more than measured in the worst case workload; a victim could be detected because its throughput with respect to the shared resource is lower than the one measured during the worst case workload.

An example of metric selection is provided in Section 4.2.

3.3 Metric characterization

An interference metric is often non-deterministic. This is due to several reasons including (but not limited to): expected interference from other tasks, non-deterministic execution order in superscalar architectures, non-deterministic execution of memory accesses in arbitrated buses and complex memory controllers, other factors depending on the architecture. In order to apply the proposed method, only the probability density function (PDF) describing the selected metric is needed.

Often the PDF should be found based on measures performed during profiling, because it could be not known before-hand. Once a suitable metric has been identified as discussed in the previous section, gathered samples should be fitted to a known PDF. If the data cannot be fitted to any known PDF, there is still the possibility of using non-parametric inference statistic to characterize the data distribution [Scholz and Stephens 1987]. An example of metric characterization is proposed in Section 4.

3.4 Thresholds determination

Once the metric has been selected and characterized as described in the previous section, the next step is to determine suitable thresholds to be used at runtime to detect interferences as they occur. Two thresholds are defined:

1. **Detection Threshold (T_D):** if the metric measured at any given time is above the detection threshold, a misbehavior is detected in the system. Depending on the observed metric, this could mean that the monitored task is either a victim or an offender.
2. **Warning Threshold (T_W):** if the metric measured at any given time is above the warning threshold but below the detection threshold, a misbehavior might be present in the system but the measure is not decisive and further observations are needed before a decision can be taken.

Both thresholds are defined starting from two confidence levels, C_D and C_W respectively, with the only constraint that $C_D < C_W$. The confidence levels can be arbitrarily decided by the designer, however he/she should consider that the values for the confidence levels define accuracy and performance overhead of the detection and recovery method.

If the confidence levels are too strict, the method can introduce a higher performance overhead, due to a larger number of false positives.
Let X be the random variable describing the selected metric, then the thresholds T_D and T_W are defined as probabilities, as in:

$$\begin{aligned} P(X \geq T_D) \leq C_D \\ P(X \geq T_W) \leq C_W \end{aligned} \tag{1}$$

From the constraint $C_D < C_W$ follows that $T_W < T_D$. Since the cumulative distribution function (CDF) of a random variable X is defined as $F_X(x) = P(X \leq x)$, the definitions can be restated in terms of CDF as:

$$\begin{aligned} F_X(T_D) \geq (1 - C_D) \\ F_X(T_W) \geq (1 - C_W) \end{aligned} \tag{2}$$

Knowing the PDF of the population from which the gathered data is extracted, i.e. knowing the behavior of the metric in a statistical way, it is possible to compute T_D and T_W numerically using the CDF of the statistics and the defined confidence levels. Please, remember that the CDF of a random variable is defined as

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda) d\lambda \tag{3}$$

where $f_X(\lambda)$ is the PDF computed in λ. From (3) follows that knowing the PDF is equivalent to knowing the CDF and vice versa. The thresholds can be determined by numerically solving (4) and (5), to find T_D and T_W, respectively.

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda) d\lambda = 1 - C_D \tag{4}$$

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda) d\lambda = 1 - C_W \tag{5}$$

Using software tools, the operations just described can be easily performed. Moreover, most data analysis tools offer statistical functions which can be used to compute T_D and T_W using their definitions.
By definition, the warning threshold identifies situations in which the metric is not within a normal range, although it is not sufficiently deviated to warrant for an immediate recovery, since the probability $P(T_W \leq X \leq T_D) = C_W - C_D$ is still high enough that $[T_W, T_D]$ is an acceptable range for the metric. However, a misbehavior can be identified if the metric values falls too many consecutive times in this range, thus we call it a *warning range*. We can define a third threshold α. We assume that the execution time of any execution of the task is independent from the execution time of the previous one by design, meaning that at the end of one execution, there are no outstanding requests to shared resources which could introduce longer delays for the next execution of the same task. Holding this assumption the probability that α consecutive executions are in the warning region while there is no interference is:

$$[P(T_W \leq X \leq T_D)]^{\alpha} = (C_W - C_D)^{\alpha} = C_{\delta}^{\alpha} \tag{6}$$

From (6) follows (7), where C_G is the confidence level that even though the metric has been in the warning region for α consecutive measurements, there is indeed no interference. The closer is C_G to 1, the higher is α.

$$\alpha = \left\lceil \frac{\ln(1 - C_G)}{\ln C_\delta} \right\rceil \quad (7)$$

It is highly likely that the data can be described by a normal probability distribution. For instance, if the fitting PDF is a normal distribution, the thresholds can be defined as in (8), (9) and (10), where $\Phi(x)$ is the CDF of the standard normal distribution, μ is the mean and σ is the standard deviation.

$$T_D = \mu + 3\sigma \quad (8)$$

$$T_W = \mu + 2\sigma \quad (9)$$

$$\alpha = \left\lceil \frac{\ln(1 - C_G)}{\ln(\Phi(3) - \Phi(2))} \right\rceil \quad (10)$$

3.5 On-line detection and recovery

The use of the proposed method requires to enable the software to detect errors induced by interferences. Once the metric has been selected and characterized, and the thresholds computed, the software should be modified to implement two functionalities:

1. **Detection:** this functionality should monitor the performance counters and signal when a violation of a detection rule is identified.
2. **Recovery:** this functionality should respond to the detection and implement the proper recovery action.

Both modules should be added to the system after the activities described in the previous sections have been completed. If possible, they should be implemented as extensions of the system scheduler.

The detection module is responsible of detecting an interference and of triggering the recovery module in the proper way. We define two detection rules:

1. **Alarm rule:** an interference is detected when a measurement finds the metric to be above the T_D threshold.
2. **Warning rule:** an interference is detected when α consecutive measurements find the metric to be within the warning range $[T_W, T_D]$.

An implementation of the detection module should perform the following actions (refer to Section 3.4 for symbols meaning):

1. Read the performance counters defined at design time to prepare the selected interference metrics, either simple or compound (Section 3.1)
2. Compare each metric to its respective thresholds T_W and T_D.
3. Trigger a recovery action if any metric is above the value of T_D.
4. If the metric is between T_W and T_D, increment a counter variable stored in the task context.
5. If such counter variable is above the threshold α, trigger a recovery action.

The recovery module should implement the recovery actions. The kind and number of recovery actions depend on the application. However, we identify two classes of recovery actions:

1. **Graceful degradation:** the system scheduling is altered in order to allow high priority tasks to perform their computations within their deadlines.
2. **Hard recovery:** the system switches to an available hot standby spare.

The designer can implement one of these recovery actions or both. For instance, an interference detected through the warning rule can be classified as non-critical. Such a violation can be recovered through graceful degradation. An interference detected

through the alarm rule can be classified as critical. Such violation can be recovered through hard recovery.

4. APPLICATION OF THE PROPOSED METHOD

The proposed method can be implemented in all tasks to identify each task either as an offender or as a victim. In this paper we implement the method in one monitored task with the objective of detecting scenarios in which the monitored task is a victim. The application and validation of the method in the case of detecting situations in which the monitored task is an offender is identical to what is described below, with the obvious exception of the metric selection.
Different kind of bugs might introduce different variations in any given metric: it is up to the designer to understand which kind of misbehavior could cause the major impact on the monitored task and define one or more suitable metrics to detect such a misbehavior. Moreover, metrics that detect a situation in which the monitored task is an offender are usually different from metrics detecting situations in which the monitored task is a victim. In this section, first we describe the benchmark application, then, repeating the structure of Section 3, we provide an example of how to implement the proposed method on the described benchmark application.

4.1 Benchmark application

The approach was evaluated using a synthetic benchmark designed to stress the memory hierarchy and to emulate the behavior of a mixed-criticality avionics application. The rationale was to evaluate the approach considering the memory hierarchy as a shared resource in conditions similar to a real use case. Since all applications must use the memory, it can be considered a very critical shared resource.

ALGORITHM 1. STRESSOR APPLICATION

```
Input: L1_line_size, data

While (1)
    for i <- 0 to L1
        for j <- 0 to length(data) step L1_line_size
            data[i+j] <- ~(data[i+j])
        end for
    end for

    wait_next_period()
end while
```

Based on this objective, we decided to create a synthetic benchmark composed of memory bounded applications. We devised two applications:

1. **Stressor:** a synthetic memory-bounded application composed of consecutive memory accesses in both read and write operations (Algorithm 1). The accesses are designed so that each of them forces the use of a new cache line. This stresses the memory hierarchy since each access to the memory forces a read of a line. After the first N accesses, where N is the number of lines in the L1 data cache, it also causes the eviction of a cache line.
2. **Bubble sort:** it performs the sorting of an integer array (Algorithm 2) [Cormen et al. 2009], sized to be bigger than the L1 data cache, in order to avoid the masking effect of that level of cache.

ALGORITHM 2. BUBBLE SORT

```
Input: unsorted; Output: sorted

while (1)
    copy(sorted, unsorted)
    do
        swapped <- 0
        for i <- 0 to length(sorted)
            for j <- i to length(sorted)
                if sorted[i] > sorted[j] then
                        swap(sorted[i], sorted[j])
                        swapped <- 1
                end if
            end for
        end for
    while (swapped)

    wait_next_period()
end while
```

The nominal system workload is composed of three bubble sort tasks and a stressor task. Henceforth, this stressor task is considered to be a high-criticality task, thus it is the monitored task. To implement offenders, a modified version of the stressor task was used. This modified version, lacks the *wait_next_period()* call at the end of the main loop. This simulates the effects of a bug, excited at any given time, causing the application to enter an infinite loop. The fact that in the stressor task each access to memory implies a cache-line eviction increases the stress on the memory hierarchy, enhancing observability.

Both applications were designed to be representative of avionic workload. Algorithm 1 is behaving like a control application. A control application typically reads data from sensors, performs some computation, and writes back results of such computation to actuators. Algorithm 2 is representative of a memory-bounded application, for instance the compression algorithm of a logging application.

4.2 Metric selection

This paragraph describes how the theory presented in Section 3.2 is applied to the defined benchmark. It also describes how a metric selection can be performed starting from the application knowledge, and how the choice can be validated by checking that the selected metric satisfies the definition of interference metric. First of all, the objective of the method has to be defined in order to select a proper metric. To define an objective, the designer should consider the tasks one at a time and define the objectives for the monitor applied to each task. Such objectives can be:

1. Identify the monitored task as an offender;
2. Identify the monitored task as a victim.

In this example, we choose a monitored task and define the objective of identifying the monitored task as a victim.

First of all, one should identify a proper metric. The definition of metric (see 3.1) requires to identify a quantity that can be measured at run time and that changes when the monitored task is affected by an interference. In this case the interference is due to other tasks (one or more) seizing access to the shared memory hierarchy, thus increasing the access latency for the victim. The selected multicore architecture does not have a metric capable of identifying directly this issue. However, the *data cache stall cycles* (DCSC) can be measured at runtime. DCSC is the number of clock cycles during which the CPU is stalled waiting for data from the cache subsystem. Given the knowledge of the benchmark application and the definition of DCSC, it is expected that

it is a good interference metric. It satisfies the first part of the definition proposed in Section 3.1, because it can be measured through performance counters. It is to be proven that it changes when the monitored task is affected by an interference. To this purpose, one can use an experimental approach.

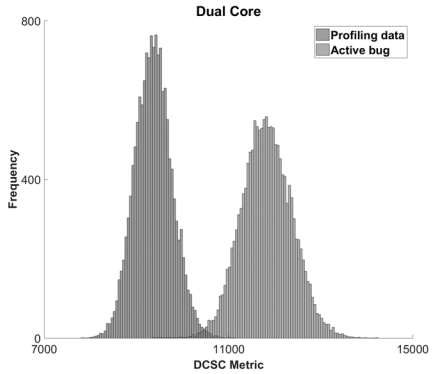

Fig. 3. DCSC metric measurements on 30,000 executions. The data labeled "*Alone*" consists of 15,000 measurements taken on the monitored task while running alone in the system. The data labeled "*With others*" consists of 15,000 measurements taken on the monitored task while running together with other tasks in the system, while each task was running its worst case workload.

During the profiling phase, the DCSC metric is measured in two scenarios:

1. When the monitored task executes alone in the system.
2. When the monitored task is running together with other tasks in the system.

The measurements performed on a dual-core architecture in such scenarios are reported in Fig. 3. The difference between the two scenarios is due to the expected interference that a task running on a multicore system experiences when other tasks are running on the same multicore system, using shared resources. In the reported scenario, the shared resource is the memory hierarchy and the gathered results show that DCSCs is an interference metric as defined in Section 3.1.

4.3 Metric characterization

In Section 3.3 we described how the selected metric should be characterized in order to be used in the method. In this paragraph, the metric selected as explained in 4.2 is characterized using the theory described in 3.3. In order to characterize the metric, the first step is to gather a sufficient number of measurements. Such a sufficient number depends on the goodness-of-fit test to be used when fitting the data to a known distribution or to a non-parametric distribution. In this example, we selected the Anderson-Darling test and its *k*-sample, non-parametric version [Stephens 1979; Scholz and Stephens 1987]. These tests are well-supported by specialized software tools, in particular in our analysis we used Mathworks' MATLAB to perform the statistical analysis described here. Since the data to be used in this characterization is the data gathered during the profiling phase, and since the requirements of the Anderson-Darling test is to have at least 8 samples, it is safe to assume that the available data satisfies this requirement. The data gathered in our experiments was fitted to a normal distribution as shown in Fig. 4. The fitting normal distribution was found, and the Anderson-Darling goodness-of-fit test with a confidence level of 0.001 was performed. The result was that the data can indeed be extracted from a population described by the fitted normal distribution. Once reached this conclusion, the designer can go on to the next phase.

However, here we show how a characterization can be reached, as a last resort, using a non-parametric distribution. Supposing that the fitted normal distribution would have failed the selected goodness-of-fit test, the designer should have looked for a different distribution. He/she could have opted for a Weibull, a Logistic or any other known PDF. If none of these could fit the data, the last resort should have been a non-parametric distribution [Gibbons and Chakraborti 2010; Corder and Foreman 2014]. Using the same software tools as before it is possible to find a non-parametric fit for the data and perform a k-samples non-parametric Anderson-Darling goodness-of-fit test. Fig. 4 shows how a non-parametric statistics based on a normal kernel fits the gathered data. Also in this case the performed Anderson-Darling test showed that the data can be extracted from a population described by the fitted statistic with a confidence level of 0.001. In the following of this section, we will use the fitted normal distribution, which simplifies from a conceptual point-of-view the application of the method, though the use of software tools makes the use of a non-parametric statistics just as simple as using a normal distribution.

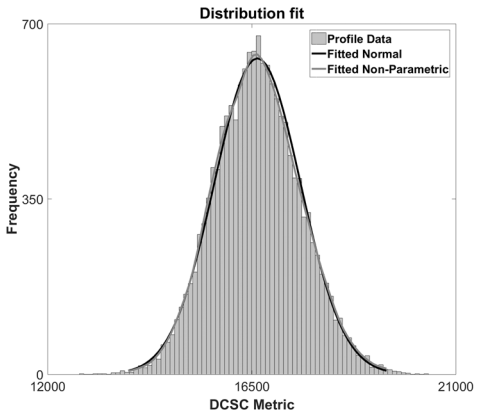

Fig. 4. Fit of the gathered data to a Normal distribution and to a non-parametric distribution. Matlab's statistics toolbox was used to fit the data to both distributions and to perform Anderson-Darling goodness-of-fit tests. The figure shows the superposition of the fitted distributions and the gathered data. The Anderson-Darling test result was that the data is modeled by the fitted normal with a confidence of 0.001. Also the non-parametric fits the data well, according to a similar Anderson-Darling test, with level of confidence 0.001.

4.4 Threshold determination

Once the best statistic to describe the data has been identified, the designer can use the gathered information to select the three thresholds described in Section 3.4. The first step is to define the confidence levels to be used in the determination of the thresholds. A good choice is to use a confidence level of $C_D = 0.05$. This confidence level defines the probability of a false positive, thus it is a good choice to keep it as small as possible; however, it also defines the accuracy of the method, with smaller confidence levels reducing the probability that an execution in which the metric is not above the threshold is actually free from interference. A good trade-off could be using a small C_D and a large C_W, thus defining a large warning range. However, this could introduce an unnecessary error latency. An interference keeping the metric in the warning range for α consecutive executions, could be detected at the first attempt with a larger C_D.
These considerations are strongly application-dependent and the designer should choose the best trade-off between accuracy and overhead, also depending on the certification requirements. Once the C_D and C_W confidence levels have been selected, it

is possible to compute the corresponding T_D and T_W thresholds using the fitted PDF found in the previous step to solve (4) and (5).

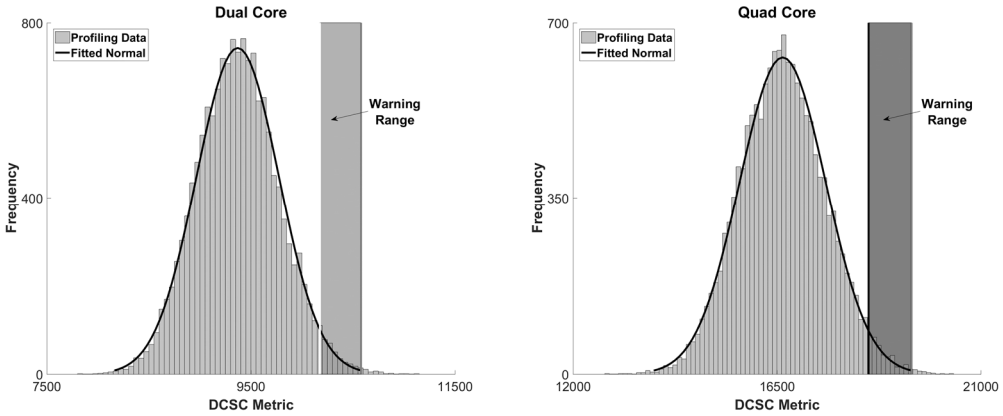

Fig. 5. Data gathered during the profiling phase, with fitted normal distribution and T_W (warning), and T_D (detection) thresholds. The warning range is delimited by the T_W threshold on the left and by the T_D threshold on the right. The graph on the left presents data gathered on the Xilinx Zynq-7000 SoC (dual core); the graph on the right presents data gathered on the NXP i.MX6Quad SoC (quad core).

In this example, we used the properties of the normal distribution and applied equations (8) and (9). Once the T_W and T_D have been computed, the warning range is defined and the α threshold can be computed using (7) or (10). The C_G confidence level defines the tolerance to be applied when the metric is in the warning range. It is by definition the probability that α consecutive measurements fall in the warning range, without an interference active in the system (see Section 3.4).
In our experiments, we decided to use a C_G close to 1. The confidence level C_G was defined to be the same in both cases. Since distributions of the DCSC were normal distributions on both the dual core and the quad core architecture, the result of the computation of α was the same (see equation (10)). Fig. 5 provides a visualization of the data, the statistical characterization, the thresholds, and the warning range.

4.5 On-line detection and recovery

The online detection and recovery was implemented as described in detail in Section 3.5. The system was implemented using a commercial type-1 hypervisor, used to partition the resources of the system between the running tasks. Since it was not possible to modify the task context, the monitoring was implemented at core level, with each core in the system running a single task. One task was selected as the monitored one, and it was instrumented to implement detection and recovery, by including the detection routine at the end of the task, before it stopped to wait for the next period, and the recovery actions were implemented as subroutines of the detection routine. The implemented recovery actions were a graceful degradation, and hot stand-by spare. Graceful degradation consisted of stopping for one period the tasks on the other cores, in order to recover functionality of the monitored task. Hot stand-by spare, was implemented as a pulse on an output signal.

5. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation performed on the proposed method. The proposed method was applied as described in Section 4 to the benchmark

application described in the same section. In this section the tools and devices used in the experiments are described, then results are presented and discussed.

5.1 Experimental Setup

The evaluation has been performed on two different multicore-based SoCs. The objective was to evaluate the approach on dual-core and quad-core architectures. To grant a significant comparison among the results, the two platforms have been selected as similar as possible. The selected platforms are the ZedBoard, based on a Xilinx Zynq-7000 [Xilinx 2015], and the SanitasEG Inventami board, based on a NXP i.MX6Quad [NXP 2015]. Both feature a SoC based on ARM Cortex-A9 architecture [ARM 2012a; ARM 2012b]. The Xilinx Zynq-7000 has a dual-core ARM Cortex-A9, whereas the NXP i.MX6Quad features a quad-core Cortex-A9. On both system, all cores have been used, and the evaluations presented in this section show how the presence of more offending tasks changes the values of the selected metric.
The required measurements where performed through an external debugger, able to observe the execution of the system and record the values of the performance counters in a non-intrusive way.

5.2 Experimental Results

Once the method was implemented as described in Section 4, a campaign of experiments was performed. The campaign consisted of running the monitored task with its worst case workload, together with other tasks. Of these other tasks, at least one was running the modified stressor algorithm, or buggy stressor, to simulate a bug causing an infinite loop with continuous accesses to the shared memory hierarchy. It is to be noticed that in this implementation, the shared memory hierarchy is not used to exchange data among the tasks, but it is nonetheless a shared resource, although each task only accesses its own memory segment. The segmentation is granted by the type-1 hypervisor.
In all experiments, the monitored task was running the stressor application (Algorithm 1). For what concerns the non-monitored tasks, we identified two classes:

1. **Buggy task:** it is a task that executes a buggy stressor application (as Algorithm 1, without the wait statement), which performs continuous accesses to the memory, both read and write, in an infinite loop.
2. **Nominal task:** it is a task that executes Algorithm 2 as expected.

The detection and recovery modules were implemented at the end of execution of the monitored task; the first action in the detection module was to read the proper performance counter in order to measure the selected metric, which was DCSC as specified in Section 4.2.
Detection and recovery overheads have not been measured, but they can be estimated. The detection overhead of the proposed approach is quite small, since it only requires that the value of a specific performance counter is read and compared to an expected value each time the detection routine is performed, which can be either at the end of each execution of the monitored task or each time the scheduler is executed. To implement the detection as described in Section 4.5, very few instructions must be added to the software. More precisely:

- One instruction to write the initial value of the relevant performance counter (maximum value minus T_D if up count, T_D if down count);
- One instruction to read the final value of the relevant performance counter

- A maximum of 3 comparisons with constant values to implement detection rules. Such constants can be compile-time constants or stored in a memory. In the latter case, the detection would also add 3 memory accesses.

This analysis allows us to state that this method has a very low detection overhead, computable as a few clock cycles.
Recovery overhead is, on the other hand, more difficult to evaluate in a general way. The cost of recovery is dependent on the adopted recovery policy, which is in turn highly application specific. However, in broad terms, the cost of recovery is either the cost of a switch to a hot standby-spare, or the loss of service from non-critical applications when a graceful degradation approach is adopted. Moreover, this loss can be temporary or permanent, depending on which flavor of graceful degradation is implemented.

5.2.1 Results on dual-core architecture

The experiments performed on the dual core architecture were performed on a Xilinx Zynq-7000 device. In these experiments there was one buggy task and one monitored task. The monitored task was executed 15,000 times, together with the buggy task. The graph reported in Fig. 6 shows the measurements taken during these experiments, together with measurements taken during the profiling phase on the same device. Table 1 reports the performed detections.

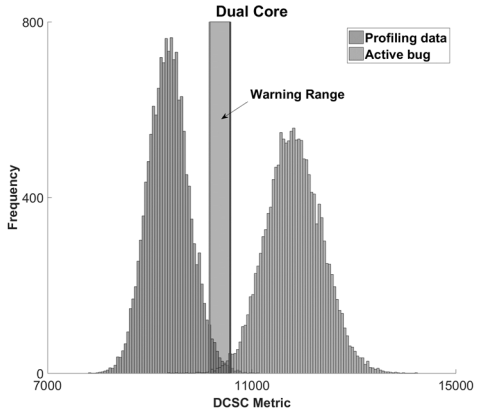

Fig. 6. Experimental results on the dual core Xilinx Zynq-7000. The graph shows the profiling data together with 15,000 measurements taken on the same monitored task while the other task was affected by a bug as described in the text. The warning range is delimited by the thresholds T_W on the left and T_D on the right.

Table 1. Detection results on the dual-core Xilinx Zynq-7000 architecture.

Alarm rule detections	*Warning rule detections*	*Tolerated*
14859 (~99.06%)	2 (~0.01%)	139 (~0.93%)

A significant number of executions triggered a detection by the alarm rule (the metric is above the threshold T_D), while a small number triggered detection by the warning rule (the metric is in the warning range for at least α consecutive times). The third column of Table 1 counts the number of executions that did not trigger any of the two rules. These executions, labeled as *tolerated*, did not cause the metric value to change

in a significant way. Thus, the temporal behavior of the monitored task was not changed enough to require a recovery action.

5.2.2 Results on a quad-core architecture

The same kind of experiments as in Section 5.2.1 were performed on a quad-core NXP i.MX6Quad-based system.
Since there are three tasks in the system besides the monitored one, there are three possible scenarios:

1. **Scenario 1:** One buggy task, two nominal tasks.
2. **Scenario 2:** Two buggy tasks, one nominal task.
3. **Scenario 3:** Three buggy tasks.

The effects on the system change significantly in the different scenarios, as reported in Table 2 and in Fig. 7.

Table 2. Detection results on the quad-core NXP i.MX6Quad architecture. Scenarios are specified in the text.

	Alarm rule detections	*Warning rule detections*	*Tolerated*
Scenario 1	1668 (11.12%)	1114 (~7.43%)	12218 (~81.45%)
Scenario 2	11348 (~75.65%)	528 (3.52%)	3124 (~20.83%)
Scenario 3	14990 (~99.93%)	0	10 (~0.07%)

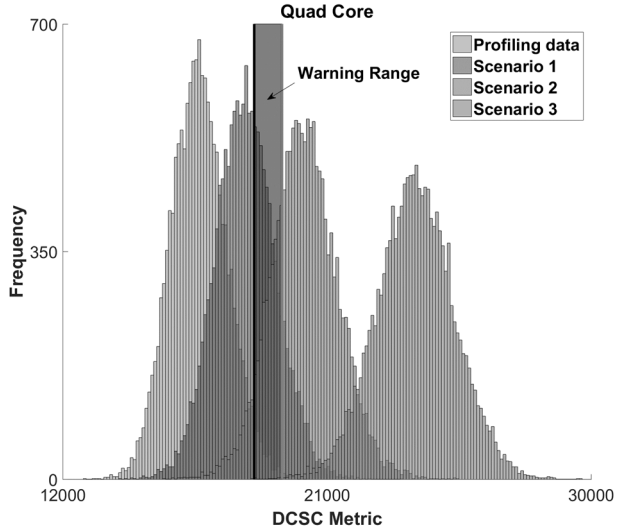

Fig. 7. Experimental results on the quad core NXP i.MX6Quad. The graph shows the profiling data together with 15,000 measurements taken on the same monitored task while other tasks were affected by a bug as described in the text. The warning range is delimited by the thresholds T_W on the left and T_D on the right. Scenarios are defined in the text.

Fig. 7 shows an overlap between the measurements gathered in the profiling phase and the measurements obtained in scenario 1. This is a hint at how the quad-core architecture is less sensitive to a misbehavior in one core, due to different architectural choices performed in the quad core processor with respect to the dual core processor. Such architectural choices are protected intellectual properties (IP) of the MPSoC vendor, and details are not available to us. However, it can be speculated that the main difference is a higher bandwidth for the bus, making it more resilient to high traffic. The data also suggests that the warning rule is most useful when there is such an overlap: the probability that in this case α consecutive measurements are in the

warning range is way higher than in a system in nominal conditions. It is also evident that a great part of executions in scenario 1 have been tolerated: this is a consequence of the lower sensitivity of the quad-core architectures to one interfering core. It is also worth noticing that in scenario 3, no detection happens due to violations of the warning rule and very few executions are tolerated: this is due to the great increment in the metric caused by the interference of three out of four tasks against the monitored task.

6. CONCLUSIONS AND FUTURE DEVELOPMENTS

This paper proposed a bounded interference approach to scheduling of multicore-based systems. The proposed method is particularly suited for mixed-criticality applications in that it allows to implement two levels of monitoring with the same analysis. The proposed method can be implemented to identify both offenders and victims. An offender is a task that is overusing a shared resource. A victim is a task that cannot use the shared resource because an offender is overusing it. By selecting proper interference metrics, the proposed method is able to detect the interference and to identify each task as either a victim or an offender. Interference is detected because an interference metric has an unexpected value. A task is identified as victim or offender depending on the metric by which the interference has been detected. For instance, a metric that measure the use of a shared resource allows identification of an offender, whereas a metric measuring throughput of a shared resource can identify a victim. Using data gathered during profiling and WCET analysis, the designer can select for each task a proper interference metric and can choose to configure the system so that each task is identified as offender, victim, or both when an interference is present. For instance, a designer can choose to reduce detection overhead in soft real-time tasks by identifying them only as offenders, whereas it can decide that hard real-time tasks should be concerned only as victims. The experiments were performed on a benchmark application in which one task has been monitored through the proposed method and identified as victim when an interference was present in the system. The benchmark application has been described and used as an example of application of the method to a mixed-criticality avionic application. The way the proposed method should be applied to an application has been described in detail and experiments have been performed on the protected benchmark application. Experiments consisted of introducing a bug causing an overuse of the memory hierarchy by one or more tasks in the system and observing that the method was able to detect the interference from a different task, which was in this case a victim. Results show that the method is able to identify significant interference and to tolerate those interferences which are not able to change the execution time of the monitored task enough to require a recovery action. Results gathered on both dual and quad core architectures show the viability of the method for what concerns interference detection and recovery. The thresholds defined in the proposed method can be fine-tuned to achieve an acceptable accuracy-performance trade-off, as needed by the designer. Such thresholds also allow to classify interferences as critical or non-critical, and implement proper and differentiated recovery actions in each case.

Future works will extend the validation of this method, by applying it to real-life multicore-based system, with special interest in mixed-criticality avionic applications. Moreover, the proposed method will be deployed in a fault-tolerant system architecture for mixed-criticality applications, to complement the detection and tolerance mechanisms already available, to get closer to the final goal of a certifiable reference architecture for avionic applications on multicore systems.

REFERENCES

J.H. Anderson, S. K. Baruah, and B.B. Brandenburg. 2009. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*. DOI:http://dx.doi.org/10.1.1.153.5608

ARM. 2012a. *Cortex-A9 Technical Reference Manual, Issue I.* ARM DDI0388I (ID091612).

ARM. 2012b. *Cortex-A9 MPCore Technical Reference Manual, Issue I.* ARM DDI0407I (ID091612).

ARM. 2014. *ARM Architecture Reference Manual, Issue C.c.* ARM DDI 0406C.c (ID051414).

Sehry Avramenko, Stefano Esposito, Massimo Violante, Marco Sozzi, Massimo Traversone, Marco Binello, Marco Terrone. 2015. An Hybrid Architecture for Consolidating Mixed Criticality Applications on Multicore Systems. In *2015 IEEE 21st International On-Line Testing Symposium*. Halkidiki: IEEE, 26–29. DOI:http://dx.doi.org/10.1109/IOLTS.2015.7229823

Sanjoy Baruah and Steve Vestal. 2008. Schedulability analysis of sporadic tasks with multiple criticality specifications. *Proc. - Euromicro Conf. Real-Time Syst.* (2008), 147–155. DOI:http://dx.doi.org/10.1109/ECRTS.2008.26

Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. 2012. Deterministic execution model on COTS hardware. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)* 7179 LNCS (2012), 98–110. DOI:http://dx.doi.org/10.1007/978-3-642-28293-5_9

Alan Burns and Robert I. Davis. 2016. *Mixed Criticality Systems - A Review* 7th ed., University of York.

Certification Authorities and Software Team. 2014. *Position Paper CAST-32, Multi-core Processors,* Federal Aviation Administration/European Aviation Safety Agency.

Gregory W. Corder and Dale I. Foreman. 2014. *Nonparametric Statistics: A Step-by-Step Approach (2nd ed.),* John Wiley & Sons, New York, NY.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rives, and Clifford Stein. 2009. *Introduction to Algorithm (3rd ed.).* MIT Press, Boston, MA.

Jean D. Gibbons and Subhabrata Chakraborti. 2010. *Nonparametri Statistical Inference (5th ed.).* CRC Press, Boca Raton, FL.

Stefano Esposito, Massimo Violante, Marco Sozzi, Marco Terrone, Massimo Traversone. 2016. Online Time Interference Detection in Mixed-Criticality Applications on Multicore Architectures using Performance Counters. *2016 IEEE 22nd International Online Testing Symposium*. To appear.

C.M. Krishna. 2014. Fault-Tolerant Scheduling in Homogeneous Real-Time Systems. *ACM Comput. Surv.* 46, 4 (2014), 48:1–48:34. DOI:http://dx.doi.org/10.1145/2534028

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. 2013. Scheduling of mixed-criticality applications on resource-sharing multicore systems. *2013 Proc. Int. Conf. Embed. Software, EMSOFT 2013* (2013). DOI:http://dx.doi.org/10.1109/EMSOFT.2013.6658595

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. 2014. Mapping mixed-criticality applications on multi-core architectures. *Des. Autom. Test Eur. Conf. Exhib. (DATE), 2014* (2014), 1–6. DOI:http://dx.doi.org/10.7873/DATE.2014.111

Marco Luise and Giorgio M. Vitetta, *Teoria dei Segnali 3/ed*, McGraw-Hill, Milano, 2009.

Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. 2014a. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. *Proc. - Euromicro Conf. Real-Time Syst.* (2014), 109–118. DOI:http://dx.doi.org/10.1109/ECRTS.2014.20

Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. 2014b. Monitoring and WCET analysis in COTS multi-core-SoC-based mixed-criticality systems. *Des. Autom. Test Eur. Conf. Exhib. (DATE), 2014* (2014), 1–5. DOI:http://dx.doi.org/10.7873/DATE.2014.080

NXP. 2014. *e6500 Core Reference Manual, Rev 0, 06/2014.* E6500RM.

NXP. 2015. *i.MX 6Dual/6Quad Applications Processor Reference Manual, Rev. 3,07/2015.* IMX6DQRM.

Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Muench, and Jan Nowotsch. 2015. Mixed-Criticality Embedded Systems -- A Balance Ensuring Partitioning and Performance. *2015 Euromicro Conf. Digit. Syst. Des.* (2015), 453–461. DOI:http://dx.doi.org/10.1109/DSD.2015.100

Rodolfo Pellizzoni et al. 2011. A predictable execution model for COTS-based embedded systems. *Real-Time Technol. Appl. - Proc.* (2011), 269–279. DOI:http://dx.doi.org/10.1109/RTAS.2011.33

J. Rushby. 1999. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance, NASA Langley Research Center, NASA CR-1999-209347

F.W. Scholz and M.A. Stephens. 1987. K-Sample Anderson-Darling Tests. *J. Am. Stat. Assoc.* 82, 399 (1987), 918–924. DOI:http://dx.doi.org/10.1080/01621459.1987.10478517

Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. 2009. Timing predictability on multi-processor systems with shared resources. *Embed. Syst. Week-Workshop Reconciling Perform. with Predict.* (2009), 87.

M.A. Stephens. 1974. EDF Statistics for Goodness of Fit and Some Comparisons. *J. Am. Stat. Assoc.* 69, 347 (1974), 730–737. DOI:http://dx.doi.org/10.1080/01621459.1974.10480196

Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *Proc. - Real-Time Syst. Symp.* (2007), 239–243. DOI:http://dx.doi.org/10.1109/RTSS.2007.47

Xilinx. 2015. *Zynq-7000 All Programmable SoC Technical Reference Manual, v1.10*. UG585.